Oasis ebSOA

An Introduction to Service Oriented Architecture

Oasis ebSOA
An Introduction to
Service Oriented Architecture

Hamid Ben Malek

Contents

List of Figures

Acronyms

AWS	Advanced Web service. An Advanced Web service is a Web service that is extended with reliability, security, and transactions capabilities
SOA	Service-Oriented Architecture. This term is an umbrella for the following items: • SOP: Service-Oriented Programming (or Service-Oriented Paradigm) • Service-Oriented Applications: software applications that spread over large geographical distances, involving many SOA Services, and different execution environments. • SOA Topology: A software architecture topology that is an alternative hybrid solution between the point-to-point and hub 'n spoke topologies. • A set of three specifications: SOA General Specification, SOA Service Specification, and SOA Fabric Specification. Like ebXML and J2EE (consisting of a set of specifications), SOA consists of three specifications.

SOA FS	SOA Fabric Specification: the SOA specification that deals with the SOA Fabric and how Service-Oriented Applications fit inside the fabric and leverage the Fabric Services in a way that is compliant with the SOA paradigm
SOA GS	SOA General Specification: the SOA specification that coordinates the two specs SOA FS and SOA SS. It is a sort of umbrella under which both specifications SOA SS and SOA FS are sitting
SOA SS	SOA Service Specification: the SOA specification that specifies how an SOA Service is constructed, how its interface is standardized, what are its contracts, schemas, the XML format of the exchanged messages, and how its interface could be dynamically discovered by other SOA Services

Introduction

SOA is the latest revolution in the IT industry. It is a tornado revolution that is still being formed, and will continue shaping itself and the industry around it for a couple of years to come (until at least 2010 where the dust may settle down for good). Why does SOA constitute a revolution and why will it take a couple of years before it is considered done, in the same way as OOP (Object-Oriented Paradigm) is considered done (that is no new innovations are being added to OOP)?

SOP (Service-Oriented Paradigm) follows on the same line the two paradigms OOP and AOP (Aspect-Oriented Paradigm). So why SOP is a revolution while OOP and AOP are not considered as big tornadoes as SOP? In other terms, why there is so much fuss about SOA than there was about OOP and AOP? The answer to this question lies on the scale SOA is targeting.

OOP, since its creation, took a long time (a couple of years) before it became the standard way of building software applications within an enterprise. Along those years, there was a little fuss about OOP, but not much as there is now about SOA. Then came AOP, which tries to complement OOP on certain software aspects that could not be elegantly dealt with using OOP (such as security that is running orthogonal to the tier architecture). The reason SOA is considered to be a big jump (as compared to what OOP and AOP introduced to the traditional old way of building software during the mainframe area), is the fact that SOA tries to cover a larger scale: a scale that spans large geographical distances, and includes enterprise applications as base components. In other terms, while OOP and AOP try both to lay down the rules of how to build a house, SOA tries to provide the rules of

how to build a city. It is the scope of SOA that is behind the fuss around it. Because the SOA scope is larger, traditional problems such as B2B and EAI (Enterprise Application Integration) could be solved within the realm of SOA. This is why SOA is considered to be a big jump compared to what OOP and AOP have accomplished.

Certain people look at SOA as being driven by the need of solving the B2B and EAI outstanding problems. But this is not entirely true. Even if B2B and EAI problems were solved entirely, SOA was destined to appear, because SOA is a natural continuation of the line containing OOP and AOP. It just happened that when SOA started to appear, the need to solve B2B and EAI problems were becoming very strong and urgent. One might say the urgent industry need to solve B2B and EAI problems has triggered the appearance of the SOA tornado, but that is just a thought with no absolute proof backing it. SOA was destined to appear regardless of B2B and EAI driven problems. It was only a question of time (when SOA was going to appear).

Microsoft had a plan to drive all their software on the .NET platform. By 2008, Microsoft Operating System together will all Microsoft applications would be completely re-written in .NET, which means that one could run Windows Operating System and Microsoft Applications on various machines such as a Sun's SPARC CPU, or a Mac CPU, not just Intel-based CPUs. Microsoft was going to deal with the SOA tornado at a later time (after 2008), in a project that aims at driving the next revolution of the Web. But since SOA tornado was triggered (by some mysterious events) and started to appear sooner than it was assumed, Microsoft shifted their initial plan toward SOA. Now Microsoft's initial plan is either changed in some way, or still exists and simply joined the SOA umbrella that has a bigger scope and aim (create the second revolution of the Web). An ESB (Enterprise Service Bus), with the code name "Indigo", is being constructed and will be part of Longhorn Operating System (may be released in 2006).

While Microsoft is driving SOA and the next revolution of the Web as part of its .NET Platform, Oasis created a TC group under the name "ebSOA" to start the SOA research and try to standardize SOA Services and SOA Fabrics. The ebSOA group may have been created as a response to the urgent industry need to solve the B2B and EAI problems, or by the ebXML folks as they are trying to upgrade their ebXML to newer versions. Whatever the cause that was behind the creation of the ebSOA group, ebSOA group should understand what is at stake here and the broader scope of its mission. Therefore, ebSOA group should not consider itself as some sort of extension to the ebXML efforts, or an extension to the Web services efforts. If the ebSOA group follows this line of thinking (an extension of ebXML and Web Services combined), the mission of ebSOA group would fail and be very scope restrictive. The mission would then fail in keeping up with Microsoft work and fail in being able to standardize the next revolution of the Web. The "eb" in ebSOA should bear no whatsoever relationship with the "eb" as in ebXML.

This paper tries to lay down the road ahead for the ebSOA group, define the concepts introduced by SOA, and find out what specifications SOA should create.

Complex solutions tend to be covered by a collection of specifications, rather than a single one. For example, the J2EE solution for building enterprise software consists of a set of specifications: EJB spec, Servlet Spec, JSP spec, JMX spec, etc... In the same way, the ebXML solution consists of a couple of specifications: ebMS, ebTA, ebCPP, etc...

Since SOA is a complex solution (targeting a much larger scale than the one covered by OOP and AOP), SOA topic cannot be covered by a single specification. Even with huge intellectual effort to make a single elegant and simple specification, such a single document would not constitute a whole picture of SOA. Therefore, there has to be a couple of specifications that together manage to paint the whole picture of SOA as one single hand. This paper tries to reduce the number of these specifications to the maximum possible. This paper suggests that there should be three specifications for SOA:

1. SOA General Specification ("SOA GS" for short): this specification coordinates the two specs "SOA FS" and "SOA SS". It is a sort of umbrella under which both specifications SOA SS and SOA FS are sitting.

2. SOA Service Specification ("SOA SS" for short): this specification specifies how an SOA Service is constructed, how its interface is standardized, what are its contracts, schemas, the XML format of the exchanged messages, and how its interface could be dynamically discovered by other SOA Services.

3. SOA Fabric Specification ("SOA FS" for short): this specification deals with the SOA Fabric and how Service-Oriented Applications fit inside the Fabric and leverage the Fabric Services in a way that is compliant with the SOA paradigm.

H. Ben Malek, Ph.D

Strategic Planning Dept,
Fujitsu Software Corp,
Sunnyvale CA 94086

Part I

SOA General Specification

1

Prelude to SOA

1.1 BRIEF OVERVIEW OF PROGRAMMING PARADIGMS

In this section we will cover briefly the history of programming languages and their paradigms. This is the first step to shed some light on the SOA technology. If SOA is the future, then showing where we come from, is somehow half the explanation of where we are going when we move toward SOA.

The evolution of a new software paradigm often progresses from programming towards design and analysis, to provide a complete path across the software development lifecycle. First, let's try to illustrate the relationship between programming languages and software paradigms:

Software design processes and programming languages mutually support each other. Design processes break a system down to smaller units. Programming languages on the other hand, have the mechanisms to define abstractions of these system sub-units, and the ability to compose those abstractions in many ways to produce the overall system. For a design process and a programming language to work well together, the programming language needs to provide abstraction and composition mechanisms that support the units the design process breaks the system into. Programming languages have a common root in that their key abstraction and composition mechanisms are rooted in a generalized procedure.

Breaking down a system into units of behavior or function is called functional decomposition. The nature of this decomposition differs between language paradigms. A software paradigm is usually associated with a family of programming languages. However some languages may support more than one paradigm.

1.1.1 Procedural paradigm

This paradigm is based on the concept of unit and scope of the data range of an executable statement. A procedural program is made of set of units or modules. Each module is composed of one or more procedures (called function, routine, subroutine, or method depending on the programming language). A procedural program may have multiple scopes where some procedures are defined inside others. Each scope can contain variables that cannot be seen in outer scopes.

1.1.2 Structured paradigm

This is a bus-discipline of procedural paradigm. This paradigm is mostly known for removing the GOTO statement. Several structuring techniques have been developed for structured programs, among them are Jackson Structured Programming and Dijkstra's Structured Programming. Some of the known structured programming languages are Pascal and Ada. Structured programming is sometimes associated with a "top-down" approach to design. Designers map out the large scale structure of a program in smaller operations, implement and test the smaller operations, and then tie them together into a system.

1.1.3 Imperative programming

This style is opposed to what is known as declarative programming. Imperative programming describes computation in terms of state and statements that change the state. It is similar to the imperative mood in natural languages when expressing commands to take action. The hardware implementation of all computers is imperative in nature (in this statement, we are excluding quantum computers which have not been released yet). From this low-level perspective, the program state is defined by the contents of memory and the instructions.

The earliest imperative languages were the machine languages of the original computers. FORTRAN, developed at IBM in 1954, was the first language to remove the obstacles of machine code in the creation of complex programs. In the late 1950s and 1960s, ALGOL was developed to allow mathematical algorithms to be easily expressed. COBOL (1960), and BASIC (1964) were attempts to make more friendly programming syntax. In the 1970s, Pascal was developed by Niklaus Wirth and C by Dennis Ritchie. Niklaus Wirth also designed Modula-2, Modula-3, and Oberon. Ada was stated in 1974 by Jean Ichbiah for the US Department of Defense, and completed in 1983.

1.1.4 Functional programming

This paradigm treats computation as the evaluation of mathematical functions. While the imperative programming emphasizes the execution of commands, functional programming emphasizes the evaluation of functional expressions which are formed using functions to combine values. Lambda calculus is the first functional programming language even if it is not executed on a computer.

Lambda calculus, designed in the 1930s by Alonzo Church, provides a formal way to describe function evaluation. In the 1950s, Newell, Shaw, and Simon at RAND Corporation, developed the first computer-based functional programming language, namely IPL (Information Processing Language). In the late 1950s, John McCarthy at MIT, developed LISP. Scheme was a later attempt to simplify and improve LISP. The language ML was created in the 1970s at the University of Edinburgh. The language Haskell was released in the 1980s to gather many ideas in functional programming research.

1.1.5 Logical programming

This paradigm is based on the logic Horn Clauses. A program consists of *facts* and *rules*. The style of logic programming is to create new statements about its model. The knowledge of the state of the world is expanded each time. Some popular application domains for logic programming are "expert systems" where the program generates a recommendation or answer from a large model of application domain, and "automated theorem proving" where the program generates novel theorems to extend some existing body of theory. Prolog and Mercury are the main languages used in this paradigm.

1.1.6 Object-oriented programming

This paradigm emphasizes the following items:

- Objects: packaging data and functionality within units. Objects are the basis of modularity and structure.
- Abstraction: the ability for a program to ignore some aspects of the information it is processing. Each object in the system plays an abstract role that can perform some work without revealing how these features are implemented.
- Encapsulation (Information Hiding): this ensures that objects cannot change the internal state of other objects in unexpected ways. Only the object's own internal methods are allowed to access its state.
- Polymorphism: the ability to refer to an object of different types, and invoking an operation on reference produces behavior depending on the actual type of the referent.

- Inheritance: the ability for objects to be defined and created as specialized types of already-existing objects.

In OOP, software is viewed in terms of the "things" (objects) it manipulates, rather than the actions it performs. Functional and procedural programmings focus on the actions rather than on the objects.

The object-oriented paradigm first took root in Simula 67, a language created by Ole-Johan Dahl and Kristen Nygaard in Oslo for making simulations. Object-oriented programming became the dominant programming methodology during the mid-1980s due to the influence of C++. Object-oriented features have been added to many languages at that time such as ADA, BASIC, Lisp, Pascal, etc... Just as procedural programming led to refinements of techniques such as structured programming, object-oriented design methods include refinements such as design patterns, design by contract, and modeling languages.

1.1.7 Component-oriented programming

There is a field in software engineering called "Software componentry". The idea is that software should be componentized (build from prefabricated components). This idea was first published in Douglas McIlroys's address at the NATO conference on software engineering (in Germany, 1968). His subsequent inclusion of pipes and filters into the Unix system was the first implementation of an infrastructure of this idea. Microsoft paved the way with OLE and COM, and nowadays several component models exist.

While in OOP, the focus is on modeling real-world interactions to create verbs and nouns which can be used in intuitive ways, software componentry makes no such assumptions and states that software should be developed by gluing prefabricated components together like in the field of electronics or mechanics. Technologies for software components include pipes and filters (for Unix), VBX, OCX/ActiveX/COM/DCOM (from Microsoft), EJB (from Sun), etc...

1.1.8 Aspect-oriented programming

AOP is a way of executing arbitrary code orthogonal to a module's primary purpose, with the intention of improving the encapsulation and reuse of the target module and the arbitrary invoked code. Refactoring or designing systems in terms of AOP is known as aspectual decomposition and is the foundation of AOP analysis.

Crosscutting could be defined as a phenomena that is observed whenever two properties being programmed must compose differently and yet be coordinated. AOP introduces *aspects* as a new modularization mechanism for separating crosscutting concerns and provides a new composition mechanism for weaving aspects back into components at well-defined *join points*. Aspects are properties of a software system that tend to cut across its main function-

ality. Synchronization, resource sharing, security, exception handling, communication, performance, memory management, and logging are examples of aspects.

Join points are elements of the component language semantics that aspects coordinate with. A static join point is a location in the structure of a component whereas a dynamic join point is a location in the execution of a component program. *Weaving* is the process of composing aspects and components related by crosscutting at the specified join points. *Aspect weaver* designates the tool that composes aspects and components.

Adaptive programming is a special case of AOP where components are expressible in terms of graphs and aspects (adaptive methods) refer to and affect the graphs using traversal strategies and adaptive visitors.

1.1.9 Event-driven programming

In this paradigm the control flow is completely controlled by external events. Event-driven programs consists of a number of event handlers that are called in response to external events, and a dispatcher which calls the event handlers, using an event queue to hold unprocessed events. Event handlers can trigger events themselves. Computer operating systems are an example of event-driven programs: interrupt handlers are event handlers for hardware events, where the CPU is the dispatcher.

1.2 TRADITIONAL ARCHITECTURE

In traditional architectures, everything was running on a monolithic machine. The user interface was a 25x80 character screen terminal for display and user input. Things were working this way for years until personal computers appear. But even so, when personal computers were in use, programs were written without thought for logical layers. Applications had their data, user interface code, and business logic code, all contained in one bundle.

Also, proprietary languages, data formats, processing data and presenting it were used. The picture in figure 1.1 illustrates such a traditional architecture:

The proprietary data access code retrieves the data in the format it understands using the language it speaks. The business logic is coded in the same bundle as the data access. The user interface is either a console or Windows form application that is tied to the application in a proprietary way.

This traditional architecture suffers from many problems:

- Functionality of the application cannot be re-used. For example, the business logic cannot be re-used because it was written specifically for this particular application and particular platform.

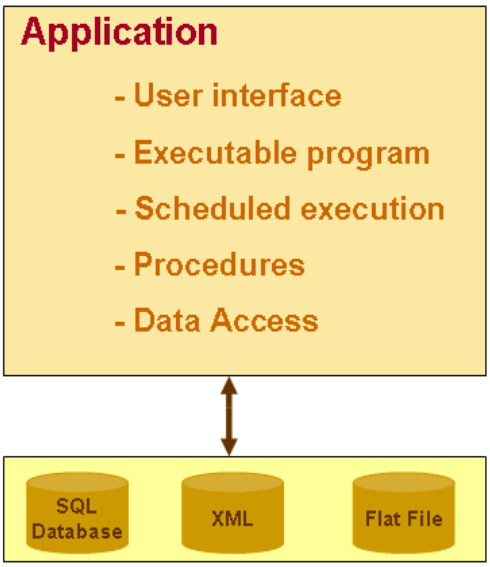

Fig. 1.1 Traditional Application Architecture

- Maintaining and debugging the application become a very complex task. Because all the code is mixed together in one bundle, any change in one part affects other parts in non-desirable way.
- Security is a major problem. The user interface cannot be separated from the application. For example, the user interface cannot be separated by a firewall.
- Integrating applications residing in different platforms becomes extremely difficult. The integrating code can only work for the applications that it was written for.
- Scalability is impossible as it is not possible to spread the application across many physical machines.

1.3 COMPONENT-BASED ARCHITECTURE

Component-based architecture was crated as applications became larger and involved more programmers and larger-scale deployments. Code re-use and breaking large applications into several pieces or components were one of the driving needs of the component-based architecture. Three pieces were universally recognized in this architecture ("universally" meaning that these pieces exist in every application):

- presentation

- business logic
- data access

The concept of functionality layers were thus introduced. It was recognized that the above three pieces should not be combined together in one single bundle as was the case in the traditional architecture. By separating these three pieces, deploying re-usable components would become a possible task. The picture in figure 1.2 illustrates the Component-based Architecture:

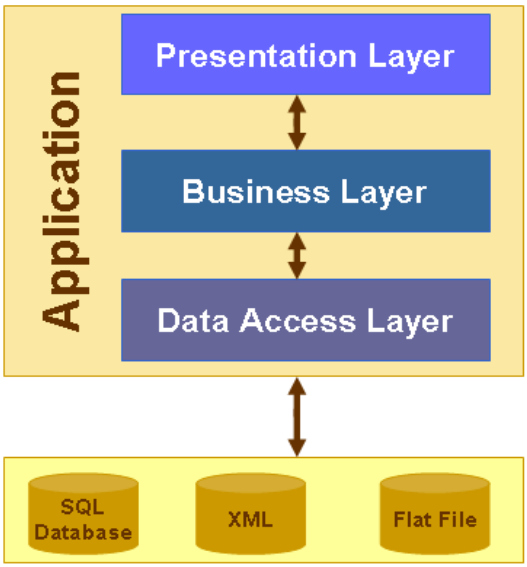

Fig. 1.2 Component-based architecture

The data access layer contains the code to connect to, query, and update the data source (be it SQL server or file system). This layer communicates with the business logic layer above it to provide it with a uniform view of the data. The presentation layer presents the processed data to the user, get instructions back, and send them back down the chain.

The component-based architecture has been used for at least a decade now with success. However, this architecture also has its own problems. Two of these problems are "component sharing" and "application integration". It is difficult to share components across different languages. Sharing components across heterogeneous platforms is even worse if not impossible. And when you add a firewall to the mix it becomes impossible for one component of one platform to call another component of a different platform. When a component-based application needs to get data from another component-based application, a user needs to read the data on one screen of the application and enter it to the other application. In case where the user interface does not exist, an integration code has to be written. This integration code is by nature specific

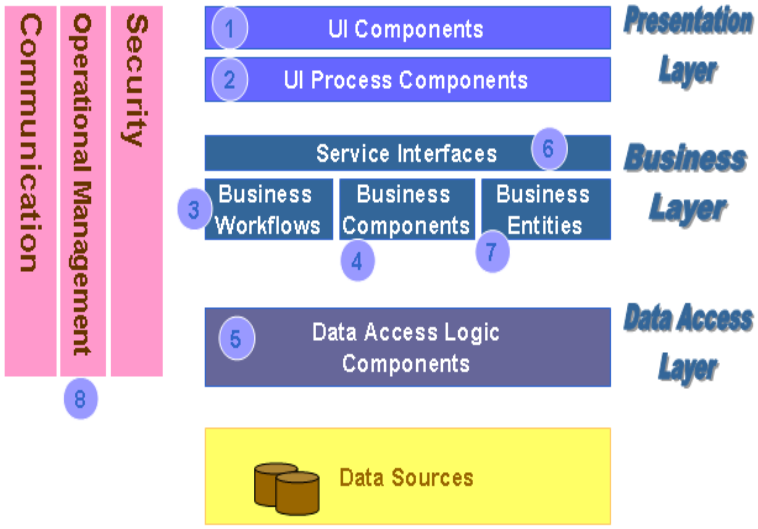

Fig. 1.3 Layered Architecture and its components

to the two applications being integrated, and is difficult to write, maintain, erratic in its behavior and any change to either application results in nothing working. This integration code creates a tightly-coupled application pair. The "Component Re-use" concept introduced by component-based architectures works well only when the components are used on the same platform and usually the same programming language that was used to write the components. Sharing business functions across heterogeneous platforms is an impossible task in component-based architecture. These shortcomings are addressed in the new architecture called "Service-Oriented Architecture".

All the software solutions that are based on a layered component architecture have several common component types. The picture of in figure 1.3 zooms into each layer to show the components that are inside it.

1. **User interface (UI) components:** Most solutions need to provide a way for users to interact with the application. For example, a Web site lets customers view products and submit orders.

2. **User process components:** In many cases, a user interaction with the system follows a predictable process. For example in e-commerce application, when a user makes a purchase, the interaction follows a predictable process of gathering data from the user, provide payment details and enters delivery details. To help synchronize and orchestrate these user interactions, it can be useful to drive the process using separate process components. This way the process flow and state management logic is not hard-coded in the user interface elements themselves and the interaction engine can then be reused b multiple user interfaces.

3. **Business workflows:** After the required data is collected by a user process, the data can be used to perform a business process. Many business processes involve multiple steps that must be performed in the correct order and orchestrated. This process could take an indeterminate amount of time to complete, so the required tasks and the data required to perform them would have to be managed. Business workflows define and coordinate long-running, multiple step business processes and can be implemented using a workflow engine.

4. **Business components:** Regardless of whether a business process consists of a single step or an orchestrated workflow, an application will probably require components that implement business rules and perform business tasks.

5. **Data access logic components:** Most applications need to access a data store at some point during a business process. It makes sense to abstract the logic necessary to access data in a separate layer of data access logic components. Doing so centralizes data access functionality and makes it easier to configure and maintain.

6. **Service interfaces:** To expose business logic as a service, an application must create service interfaces that support the communication contracts its consumers require. Service interfaces are also referred to as *business facades*.

7. **Business entity components:** Most applications require data to be passed between components. The data is used to represent real-world business entities, such as products and orders.

8. **Components for security, operational management, and communication:** An application will probably also use components to perform exception management, to authorize users to perform certain tasks, and to communicate with other services and applications. In figure 1.3 these components are drawn vertically to illustrate that they are cross-cutting aspects (they cross cut the three layers, and hence are orthogonal to them).

2

Service-Oriented Architecture

Service architecture is not only about building SOA Services, but also about building SOA applications by composing and combining various SOA Services around the globe, and leveraging the SOA Fabric Services. If we look at Service-Oriented Architecture from the smallest possible scale (namely the enterprise scale), the architecture becomes what we call an SOA Service. In other terms, the smallest SOA application is an application that is composed of one SOA Service only. Component-based architecture could be compared to an SOA Service (the smallest SOA application possible) because the two share the same scale, namely the enterprise scale. One cannot compare a component-based application to a general SOA application that is made of more than one SOA Service, because the two applications do not have the same scale.

2.1 SERVICE ARCHITECTURE

In this section we will look at the "Service-Oriented Architecture" from the smallest scale possible, namely the enterprise scale (not the inter-enterprise scale), and compare it with the "Component-based Architecture".

As the reader may recall in the previous chapter, a component-based architecture introduced three layers: presentation, business logic, and data access layers. The service-oriented architecture makes certain modifications to this architecture:

- First, it eliminates the presentation layer.

- Second, it creates a new layer called "Service Layer" or "Service Interface" (or "Service Definition"), that sits on top of the business layer, as the presentation layer did in the component-based architecture.

The reader may wonder why the presentation layer disappeared in the service-oriented architecture and what happened to it. Is a service-oriented architecture blind, without eyes, since it seems that it does not have a presentation view? We will get to this question later. The picture in figure 2.1 illustrates the smallest SOA application, that is an SOA Service:

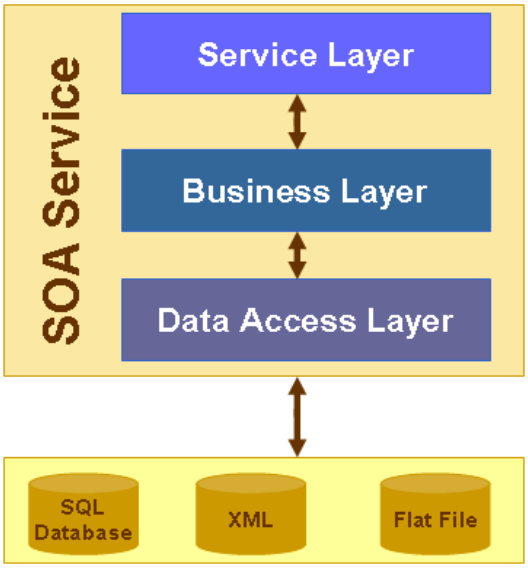

Fig. 2.1 Service-centric architecture

The particular case where the "Service Layer" describes its interface in terms of SOAP and WSDL, we get a particular case of an SOA Service known as "Web service". A general SOA Service need not be a Web service. That is, the interface of the "Service Layer" need not necessary be described in SOAP and WSDL. Web services are just very particular cases of SOA Services[1]. Another distinction that the reader needs to understand between an SOA Service in general and a Web service, is that the term Web service applies only to the "Service Layer" (when it is described in SOAP and WSDL), whereas the term "SOA Service" applies to the whole enterprise application, namely the three layers: Service Layer, Business Layer, and Data Access Layer.

[1]Technically speaking, an SOA Service, as defined later in this chapter, is a more general type of service, whereas a Web service is not exaclty an SOA Service but it could be made an SOA Service by adding some patches

By decoupling the presentation layer from the business layer, more functionality can be added to the service interface such as security, routing, and transactions.

What is so special about the "Service Layer"? While the "Business Layer" and "Data Access Layer" both varies from one enterprise application to another, the "Service Layer" is almost identical between two different enterprise applications. This is because the service layer contains the following common items:

- Common encapsulation
- common security
- common alphabet, language, and format
- common error handling

The service layer being the "same" from one application to another is what makes the interoperating possible between two heterogeneous platforms. The picture in figure 2.2 zooms into the "Service Layer":

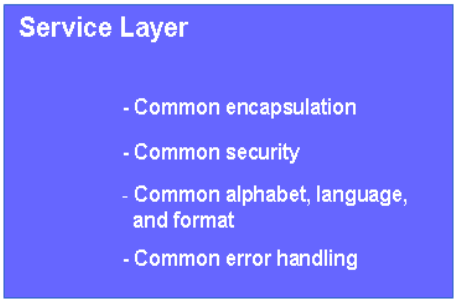

Fig. 2.2 Service Layer

To answer the question of what happened to the "Presentation Layer" in Service-Oriented Architectures, here is the reason why the presentation layer was eliminated from the service architecture: A service definition does not have a presentation layer because it is intended to have many presentation layers, not just one. Because the "Service Layer" exposes a universal interface, it can be used and called by any user interface that resides on any platform. In component-based architectures, the presentation layer was an integral part of an enterprise application. That is, even though the presentation layer (i.e. user interface) is decoupled from the business logic and may even run on a different machine as a separate tier, the fact is that it is considered part of the enterprise application. This is not the case in service architectures because the user interface is not so coupled to the business interface. In service architectures, the business layer is "protected", that is wrapped by a layer (the service layer) that exposes a universal interface to be used by any

user interface (presentation layer) on any platform. The picture in figure 2.3 shows an SOA Service having multiple user interfaces.

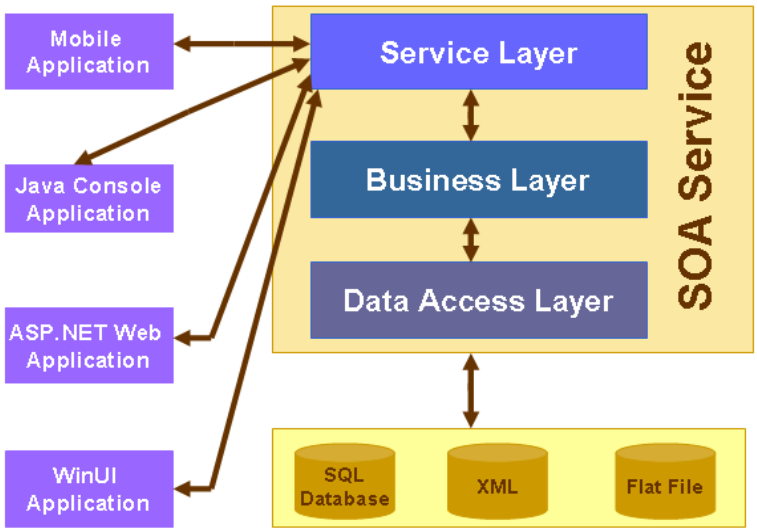

Fig. 2.3 An SOA Service with multiple presentation layers (user interfaces)

2.2 SOA SERVICE

First, we start by providing a precise definition of what an SOA Service is: An SOA Service is an enterprise application that satisfies the following properties:

- It is autonomous
- It exposes an interface through a standard XML-based contract, that one interacts with via message exchanges. These messages are XML-based, and could be embedded/transported via different protocols, including SOAP/HTTP, SOAP/TCP, XML/HTTP, XML/TCP, XML/SMTP, SOAP/SMTP, etc...
- It is stable and always available
- Compatibility between SOA Services is based only on policy
- It has an explicit boundary
- It shares schema and contract (not libraries) with other SOA Services

In practice, an SOA Service is built by modifying an existing component-based enterprise application: decoupling the presentation layer from the business layer so that the presentation layer could be detached and completely

eliminated. Then wrapping the business layer by writing a new layer (the "Service Layer") that exposes an interface satisfying the properties described in the definition above.

2.2.1 Contract and Schema

2.2.2 Policy

3

SOA Fabric

4

SOA Patterns

Part II

SOA Service Specification

5

SOA Service

5.1 DEFINITION

5.2 SERVICE INTERFACE

5.2.0.1 Contract and Schema

5.2.0.2 Policy

Part III

SOA Fabric Specification

6

Fabric Generic Services

Appendix A Oasis ebSOA Technical Committee

The following persons are members of the Oasis ebSOA Technical Committee, and are listed alphabetically ordered by their first name:

Table A.1 Oasis ebSOA TC.

Anders Tell	Individual	Member
Bernd Eckenfels	Seeburger, AG	Member
Dale Moberg	Cyclone Commerce	Member
Dan Pattyn	Individual	Member
David Burdett	CommerceOne	Member
David Layton	General Services Administration	Member
David Webber	Individual	Member
Duane Nickull	Adobe Systems	TC Chair
Ed Chase	Adobe Systems	Member
Hamid Ben Malek	Fujitsu Software Corporation	Member
Ian Jones	BTplc	Member
Jeff Turpin	Cyclone Commerce	Member
John Aerts	LA County Information Systems Advisory Body	Member
John Yunker	Individual	Member
Joseph Chiusano	Booz Allen Hamilton	Member
Kathryn Breininger	The Boeing Company	Member
Kishor Kalivarapu	CommerceOne	Member
Matthew MacKenzie	Adobe Systems	Secretary
Neelakantan Kartha	Sterling Commerce	Member
Nita Sharma	Individual	Member
Paul Spencer	Individual	Member
Puay Siew Tan	Singapore Institute of Manufacturing Technology	Member
Ravi Panj	Individual	Member
Ron Schuldt	Lockheed Martin	Member
Sally St. Amand	Individual	Member
Steve Ross-Talbot	Enigmatec Corporation Ltd	Member
Steve Capell	Individual	Member
T Mathews	LMI Government Consulting	Member

Glossary

AWS Advance Web service. An Advance Web service is a web service that is extended with reliability, security, and transactions capabilities.

ESB Enterprise Service Bus. An Enterprise Service Bus is a special type of **Fabric**. It is made of a collection of interoperating Advanced Web Services that are spread over a large geographical distance. An Advanced Web service is a Web service that includes functionalities such reliability, security and transactions.

Execution Environment This term is almost synonymous with "Platform". It refers to the type of machines, the operating system, software applications and tools deployed on top of the operating system. The term "platform" usually refers only to the machine types being used and the operating system being run on the machines.

Generics In computer science, generics are a technique allowing a value to take different datatypes as long as certain contracts (called subtype) are kept. OOP languages, C#, C++, D, BETA, Eiffel, Ada, and a later version of Java provide generic facility.

Paradigm It comes from the Greek word "paradeigma" meaning pattern or example. The Greek word "paradeiknunai" means demonstrate. From the late 1800 the word paradigm has been used as an epistemological term to mean "thought pattern".

Policy Expressions Policy expressions, also known as Policy Assertions, indicate which conditions and guarantees must hold true to enable the normal operation of the service.

Programming paradigm It is *paradigm* for software system development. It gives the view a programmer has of the execution of the program. For example, in the case of "Object-Oriented Programming", a programmer sees the execution of the program as a collection of interacting objects. A *programming paradigm* is usually connected to a school of software architecture, and is associated with a family of programming languages. For example, Java and Smalltalk are associated with "Object-Oriented Programming", while Haskell and Scheme are associated with "Functional Programming". The following is a list of existing programming paradigms/styles:

- Structured programming (sub-discipline of procedural programming)
- Unstructured programming
- Imperative programming (style)
- Declarative programming (paradigm. Combines both functional programming and logic programming)
- Generic programming (style)
- Procedural programming (paradigm)
- Functional programming (paradigm)
- Object-oriented programming (paradigm)
- Component-oriented programming
- Logical programming (paradigm)
- Aspect-oriented programming (paradigm)
- Post-object programming (an extension to object-oriented programming)
- Relational programming
- Symbolic programming
- Ars based programming
- Event-driven programming (paradigm)
- Intentional programming (paradigm)
- Subject-oriented programming
- Service-oriented programming (paradigm)

Semantic Compatibility Compatibility is a concept that indicates whether two given SOA Services can interoperate (are compatible). There are two types of compatibility: semantic and structural. Semantic Compatibility is

based on explicit statements of capabilities and requirements in the form of a policy.

Service Domain A collection of SOA Services sharing common services such as security. For a client application to contact an SOA Service within a Service Domain, call needs first to go through the Service Domain. A Service Domain is like a container hosting SOA Services. If an SOA Service could be compared to an EJB, a Service Domain would then correspond to an EJB Container.

SOA Service An autonomous software application exposing an interface through a standard XML-based contract, that one interacts with via message exchanges. These messages are XML-based, and could be embedded/transported via different protocols, including SOAP/HTTP, SOAP/TCP, XML/HTTP, XML/TCP, XML/SMTP, SOAP/SMTP, etc... An SOA Service should be stable and always available. Compatibility between SOA Services is based only on policy.

Service-Oriented Application A software application containing SOA Services spread over large geographical distances, multiple trust authorities, and distinct execution environments.

SOA Service-Oriented Architecture. This term is an umbrella for the following items:

- SOP: Service-Oriented Programming (or Service-Oriented Paradigm)
- Service-Oriented Applications: software applications that spread over large geographical distances, involving many SOA Services, and different execution environments.
- SOA Topology: A software architecture topology that is an alternative hybrid solution between the point-to-point and hub 'n spoke topologies.
- A set of three specifications: SOA General Specification, SOA Service Specification, and SOA Fabric Specification. Like ebXML and J2EE (consisting of a set of specifications), SOA consists of three specifications.

SOA Fabric (or Fabric for short) A particular kind of Service-Oriented System that plays the role of infrastructure in which SOA applications can run. An ESB (Enterprise Service Bus) is a special kind of Fabric (a Fabric that is made of collaborating set of Advanced Web Services). A Fabric provides a set of SOA Services such as an SOA Discovery Service (similar to ebXML registry and Web Services UUDI), a Context Service, an Activity Service, a Coordination Service, and other concepts such as Orchestration Language and Service Domain. A Fabric need not necessarily be implemented as a set of collaborating Advanced Web services.

SOA FS SOA Fabric Specification: the SOA specification that deals with the SOA Fabric and how Service-Oriented Applications fit inside the fabric

and leverage the Fabric Services in a way that is compliant with the SOA paradigm.

SOA GS SOA General Specification: the SOA specification that coordinates the two specs SOA FS and SOA SS. It is a sort of umbrella under which both specifications SOA SS and SOA FS are sitting.

SOA SS SOA Service Specification: the SOA specification that specifies how an SOA Service is constructed, how its interface is standardized, what are its contracts, schemas, the XML format of the exchanged messages, and how its interface could be dynamically discovered by other SOA Services.

Service-Oriented System A collection of deployed SOA Services interoperating with each others.

SOP Service-Oriented Programming (or Service-Oriented Paradigm). SOP is a continuation of the historical evolution of Programming Paradigms. Programming paradigms are theoretical concepts with concrete techniques on how software programs and applications should be built. SOP is the last item in the line including OOP (Object-Oriented Programming), and AOP (Aspect-Oriented Programming). While OOP, and AOP cover building software at the enterprise level only, SOP deals with building software at a larger scale, usually a scale covering large geographical distances with many other enterprise applications as base components. OOP and AOP are used for software that is tested, deployed, and versioned as an atomic unit (usually within the same organization and involving only the intranet of the organization). SOP is used for integrating autonomous applications that are tested, deployed and versioned independently. If software could be compared to buildings, then OOP and AOP are complementing rules for building a house, while SOP provides the rules for building a city.

Structural Compatibility This compatibility is based on contract and schema (and it can be validated if not enforced).